I/O Systems

Device Management — objectives

abstraction from details of physical devices

uniform naming that does not depend on hardware details

serialization of I/O operations by concurrent applications

protection of standard-devices against unauthorized accesses

buffering if data from/to device cannot be stored in final destination

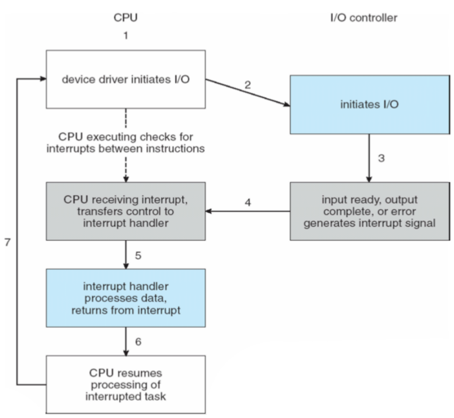
error handling of sporadic device errors

virtualizing physical devices via memory + time multiplexing

Device Management — techniques

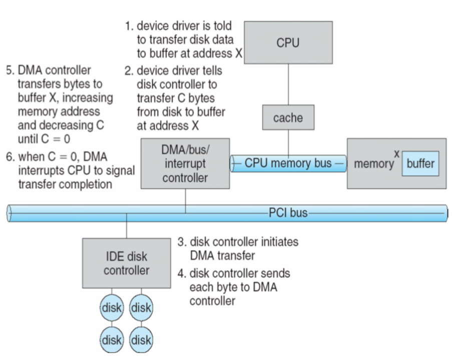
programmed I/O:
- thread is busy-waiting for I/O operation to complete ⟶ CPU cannot be used elsewhere
- kernel is *polling* state of I/O device (command-ready, busy, error)

interrupt-driven I/O:
- I/O command is issued
- processor continues executing instructions
- I/O device sends interrupt when command is done



direct memory access (DMA):
- DMA module controls exchange of data between main memory and I/O device
- processor interrupted after entire block has been transferred

⟶ bypasses CPU to transfer data directly between I/O device and memory



Kernel I/O Subsystem

scheduling: order I/O requests in per-device queues

buffering: store data in memory while transferring between devices

error handling: recover from read/availability/write errors

protection: protect from accidental/purposeful disruptions

spooling: hold output to device if device is slow (e.g., printer)

reservation: provide exclusive access for process

Device Drivers

jobs:
- *translate* user request through device-independent standard interface
- *initialize* hardware at boot time
- *shut down* hardware

Device Buffering

reasons:
- without buffering threads must wait for I/O to complete before proceeding
- pages must remain in main memory during physical I/O

version 1 — block-oriented:
- information is stored in fixed-size blocks
- transfers are made a block at a time
- used for disks/tapes

version 2 — stream-oriented:
- transfer information as byte stream
- used for keyboard, terminals, … (most things that is not secondary storage)

Buffering — user level

principle: task specifies memory buffer where incoming data is placed

issues:
- what happens if buffer is currently paged out to disk? ⟶ data loss
- additional problems with writing? ⟶ when is buffer available for re-use?

Buffering — single

principle: user process can process one data block while next block is read in

swapping: can occur since input is taking place in system memory, not user memory

stream-oriented: buffer = input line, carriage return signals end of line

block-oriented:
- input transfers made to *system buffer*
- buffer moved to *user space* when needed
- another block read into system buffer

Buffering — double

principle: use 2 system buffers instead of 1 (per user process)

user process can write/read from one buffer while OS empties/fills other buffer

Buffering — circular

problem: double buffer insufficient for high-burst traffic situations:
- many writes between long periods of computations
- long computation periods while receiving data
- might want to read ahead more than just single block from disk